Managing Schema Changes in Object-Relationship Databases*

Mohamed Bouneffa and Nacer Boudjlida

Centre de Recherche en Informatique de Nancy (CRIN-CNRS)
Campus scientifique B.P. 239 54506 Vandoeuvre-lès-Nancy

Abstract. Changing a database schema may affect the preexistent data instances and programs. Two major strategies has been followed to deal with schema changes side effects on data and programs. The first, called data restructuring consists in converting data to render them conform to the modified schema. However it doesn't concern the programs. The second approach allows the coexistence of several schema versions into the same database. Such an approach uses the emulation mechanism to achieve programs conformity to the new schema. However, it introduces data redundancy and both restructuring and emulation only consider atomic changes.

We introduce an approach combining both databases restructuring and emulation mechanisms. We consider the data instances as a set of multiple facets, each one representing according to a particular schema version. To avoid data redundancy and to achieve programs adaptation to schema change, only one facet is physically stored. In fact we define mapping functions allowing every program defined for a particular facet to be performed even if this facet is not effectively stored. Our approach is applicable for both atomic and complex schema changes.

1 Introduction

Database systems are built to support long-term persistent data. So, it is natural to expect that, over time, data management needs will change and drawbacks will be highlighted. In order to satisfy these needs the database schema must be changed. Indeed, the database needs to be restructured to meet the modified schema. At the same time, existing programs may also need to be modified to meet the new database structure. In some areas, like engineering database, the management of several data schema versions is also needed. For these reasons, support for *full schema evolution* is a required facility in any DB system. Full schema evolution concerns schema change or modification, instance adaptation to these changes and schema versioning. This paper is devoted to the study of database adaptation and schema versioning as a side effect of schema modification. We consider both atomic schema changes like attribute deletion or modification, and composite operations as those performed in the schema integration activity[?, ?].

* In proceedings of the 14th Int. Conf. on Entity-Relationship and Objected-Oriented Modelling, OO-ER'95, pages 113-122, December 1995, Bond Univ. Gold Coast, Queensland, Australia

We propose an approach permitting both instances and programs adaptation. The approach is based on multi-facetted data defined as homomorphic functions. Each facet represents the state of an instance in a particular schema version. Relationships between the facets are then defined and used to generate mapping functions allowing programs compatibility.

The remainder of this paper is structured as follows: we review some of relevant approaches dealing with the problem of instances and programs adaptation. We introduce then, in the third section, the Object-Relationship data model used as a support of our work. The fourth section defines the facet mechanism and shows its use in schema evolution. We end the paper by implementation considerations and concluding remarks.

2 Related work

We present five approaches of instances and/or programs adaptation that we consider significant in the field. These are : *conversion*, *emulation*, *polymorphism*, *temporal and historical models* and *dynamic instances classification.*

Instance *conversion* was introduced by *Gemstone* [12] and *Orion* systems. In *Gemstone*, all the affected instances must be restructured *"off line"*, as a result of a DBMS administrator request. This leads the database to be unusable during the restructuring time. In contrast, *Orion* allows data restructuring "*on line*" by hiding deleted attributes or enlarging the bounds of a newly generalised domain. In fact, schema change operations allowed by *Orion* do not affect the already stored data, except attributes deletion. Thus, *Orion* does not provide user-defined conversion procedures, limiting then, schema changes to a restrictive predefined list[3]. The conversion as used in both *Orion* or *Gemstone* does not allow programs compatibility : there is in fact, no explicit relationship between different versions of a schema [7].

Emulation[14] provides schema versioning and programs compatibility by means of a class versioning scheme. Every class change produce s a new version of this class. Each instance or program is then, related to a particular class version. Mapping between an instance of a new class version and methods defined in the old ones is performed by a set of system routines implemented as exception handlers. Like conversion, emulation only deals with a restrictive and predefined list of atomic class changes.

The use of the polymorphism. Polymorphism has been used to achieve program compatibility without data instances restructuring. In [11], a polymorphic algebra was defined by a set of operators which can be executed under different representations of the same piece of data. Polymorphic algebra is more interesting to support transformations from atomic structure to an aggregation one, or to a set, and from an aggregation to a set. However, it can't be used in the case of class or attribute deletion.

Temporal and historical models often consist in an extension of the relational data model to manage the time as an attribute associated with the real world entities[2, 9]. Thus, time can be used as a relevant parameter in queries. This approach only allows data retrieval in several versions of a schema. It does not deal with data sharing problems between schema versions nor with programs compatibility. Thus, data redundancy may occur and old programs must be explicitly transformed by introducing the convenient time values in the old DB queries.

Dynamic classification has been introduced in [10] to support incremental database schema design. In this approach an instance is dynamically related to a class and each class change can make instances migrate to other more relevant classes. Such an approach doesn't deal with schema versioning nor programs compatibility. In fact, managing several versions of a DB schema in presence of instances migration is a very hard problem. Since, generally, data instances are referenced by their classes, then the migration of an instance leads to update all the relationships in which the instance participates.

2.1 Discussion

The approaches we have presented may be considered as special cases of two more general strategies, dealing with the management of schema changes side effects on databases. The first strategy consists in restructuring the database while the second allows the coexistence of multiple schemas for a single database.

Following the database restructuring strategy, data instances must be converted to be conform to the modified schema. Such a conversion can be made by some atomic operations or by more complex ones like the dynamic classification, atomic domains to tuples conversions, etc.

Restructuring a database leads, naturally, to transform the affected programs operating on its instances. These transformations have rarely been considered in the studied approaches. In [15] a method using λ-calculus has been developed to compare two behavioural components (object methods) of two distinct OODB schemas. Such a comparison gives a significant knowledge about what must be done in order to adapt a method defined for one schema version to another one. Using the polymorphism of DB programs is an alternative way to achieve programs adaptation to schema changes. However, polymorphism can only be used for a limited set of changes.

Another important question that should be addressed, with databases restructuring strategy, is how to share the data instances between all the schema versions? In *Orion* a data instance can be shared by several schema versions, avoiding then data redundancy. However, this sharing is possible because the schema changes considered are quite simple and do not require an effective restructuring. To reduce data redundancy, some approaches [1] limit schema versioning to only specific parts of a database schema.

Emulation is a special case of the strategy allowing the coexistence of multiple schema in the same database. This strategy tries to avoid the database restructuring and to provide schema versioning by associating each data instance and each program to a specific schema version. Such a strategy increases the data redundancy and may, then, cause some semantical problems in real world data interpreting. In fact, there is no relationship between two pieces of data representing the same real object into two distinct schema versions.

To avoid inconvenient of both databases restructuring and the coexistence of multiple schemas, we define an approach combining these two strategies. Our approach allows each piece of data to be viewed as an instance of more than one DB schema. To do that, data instances are multifaceted and each facet corresponds to one DB schema. Since only one facet may be physically stored, all the facets are related by means of mapping functions. Such functions are used by the programs operating on them to achieve programs compatibility.

The following section introduces the data model underlying to our approach.

3 The GORM data model

GORM (Generic Object-Relationship Data Model) has been defined in order to manage complex data generally used in advanced database environments, like CASE databases[4]. GORM has been defined as a synthetic model of the more relevant semantic and object oriented data models. GORM encompasses basic types of objects as well as classical semantic constructors and relationships among objects. The concepts of the data model are briefly exposed.

3.1 Object types

There are three kinds of object types:

Basic types are generally used for Input/Output operations. Such types may be atomic (integers, boolean, string, ...) or structured like arrays, lists or tuples. No identifier is associated with basic types values.

Entity types have to be interpreted as entities in the Entity/Relationship data model[?] or as abstract types in semantic data models [?]. Each entity must be identified by an internal identifier.

Relationship types represent semantical associations between entity types (e.g. the relationship type $Module$ in the right part of figure 1). The two entity types associated by a relationship type are called *participants* (e.g. $Specification$ and $Body$) and are labelled by string values representing their respective roles (e.g. *sp* and *bod*) and a cardinality which may be single (single arrow for one-to-one) or multiple (double arrow for one to many). Relationships are also identified by internal symbols.

3.2 Relations among objects

There are three additional means to link different data types.

The attribute relationship can be viewed as a function $a : T_1 \longrightarrow T_2$ where T_1 may be an entity or relationship type and T_2 an entity or basic type. When T_1 is a relationship type, T_2 may only be a basic type. T_1, T_2 and a are respectively called domain, range and name of the attribute.

Composition relationships or links defined between two entity types can be viewed as a combination of classical aggregations (tuples) and grouping. The figure 1 shows a module as an aggregation of a specification and many bodies (or implementations). A composition link is defined giving a composite entity type (e.g. *Module*) and the components (e.g. *Specification* and *Body*) which are designated by their roles (*spec* and *bodies*) and their cardinalities.

Subtyping relationship among two entity types is the classical IS_A relationship restricted to single inheritance.

The following section introduces the notion of multiple facets associated to an instance.

4 The use of multiple facets in schema evolution

We propose to promote data sharing between several schema versions and programs compatibility by considering every instance i of a piece of data as a sequence of facets $\{f_1(i), \ldots, f_n(i)\}$. Each facet $f_j(i)$ represents the value of the instance i in a version V_j of the DB schema. We first give an example that, intuitively, introduces the notion of facets and then present these facets in a formal way. Finally we show how these facets must be used in presence of schema changes.

4.1 Informal presentation

Figure 1 shows two versions of the entity type $Module$. Version V_2 of the schema results from changes performed on version V_1 :

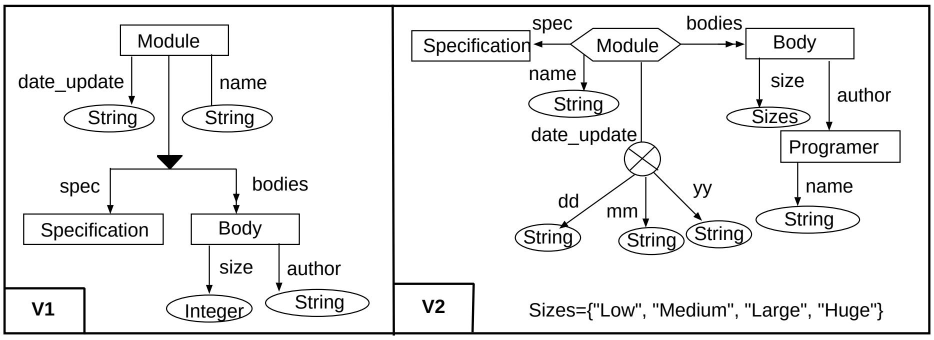

Fig. 1. Example of schema changes

- the module's attribute *name* has the same name and domain in V_1 and V_2. So, it can be shared by the two versions without any database restructuring,
- the entity type $Module$ has been transformed into a relationship type with, as its participants, the entity types $Specification$ and $Body$,
- in V_1, the attribute *Body.programmer* is a string value representing a programmer's name. It is represented, in V_2, by an entity type $Programmer$. The attribute *Body.programmer* as defined in V_1 corresponds to the new defined attribute $Programmer.name$,
- the domain of the attribute $Module.date\_update$ is restructured from a string value (in V_1) into a more complex value (aggregation or tuple) in V_2. Further, the domain of $Body.size$ is changed from integer in $V1$ to the set $\{"Low", "Medium", "Large",$
  $"Huge"\}$ in V_2.

Let m, b and p be instance identifiers of $Module$, $Body$ and $Programmer$ respectively (figure 2). m and b could then be represented by two facets f_1 and f_2 according to the two schema versions V_1 and V_2.

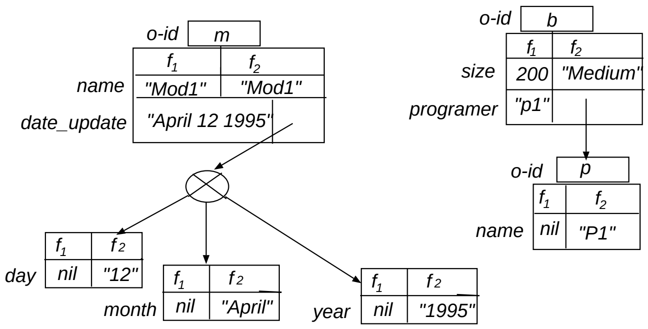

Fig. 2. Multifaceted instances

1. f_1 and f_2 share the same entity identifiers and all unchanged properties (attributes and components). For instance, they share the same value for the attribute $Module.name$ and the same identifiers of b and m,
2. the respective domains of $Body.size$ in V_1 and V_2 are different, but it is possible to define an explicit function f that converts integer values of $Module.size$ as defined in V_1 to their corresponding string values in V_2,
   This case shows the impossibility of defining mapping functions between two facets for every schema change. It is, in fact, impossible to retrieve the exact size of a module by just knowing that it belongs to the huge or small programs categories. A detailed deal with such problems can be found in [8, 13],
3. the respective values of $Module.date\_update$ in V_1 and V_2 differ by their structures. A mapping between these two values can be performed by aggregate, disaggregate functions which transform an atomic value to a tuple and vice versa [5],
4. the attribute $Body.programmer$ as defined in V_1 is replaced by the identifier of an instance b of the type $programmer$.
5. attributes which are defined in only one schema version have an undefined value for the other version. For instance the $Module.date\_update.year$ have an undefined value in V_1.

Informally, our approach consists in associating a set of facets with every database (one facet per schema version). Such facets share the identifier of the instances and some attribute values (unmodified ones). For the modified properties, mapping functions may be defined as gateways between the different schema versions. Mapping functions may be used to achieve both data conversion and programs compatibility. Such functions may be automatically generated during the schema evolution process or may be user defined. The following section, defines the notion of multiple facets and mapping functions in a more formal way. We also explain the strategies which may be followed to achieve schema reuse and data and programs adaptation. We then illustrate the application of the approach to the most relevant schema changes.

4.2 Formal definition of multiple facets

Over time, a database D may be viewed as a sequence of schemas S_i and database states D_i^j where every state is conform to the corresponding schema S_i (figure 3). D_i^j transitions are performed through database updates while S_i transitions re-

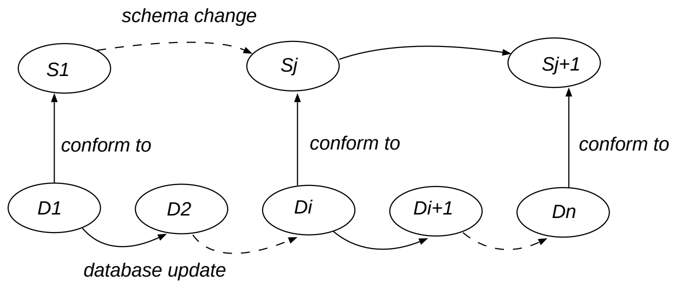

Fig. 3. Database and schema evolution

sult from schema changes. Afirst problem is how to make a database state D_i^j conform to a database schema S_{i+1}? A data item that has different definitions in the schema versions is associated with a function that maps its value according to its definition in a schema version into its value according to its definition in another schema version. An item who has the same definition in the schema versions has the identity function associated with it. Thus, mapping functions help in restructuring D_i^j into a database D_{i+1}^j conform to the new schema version. An additional usefulness of the mapping functions is in program restructuring. Indeed, some functions can be used to re-write method invocations (ie. operations that are addressed to the objects in the database) while others can be used to make some methods polymorphic. Instead of storing every facet, to avoid data redundancy data may be stored following a synthetic schema version. Thus, mapping functions will be used to derive a data value for a given schema version from the synthetic one.

Multiple facets are defined only for entity and relationship types instances. Atomic and constructed values are "mandatorily" related to entities (respectively relationships) by means of attribute links, so their facets are defined inside the entities (respectively the relationships).

An entity e is formally defined as a tuple $[id;\ vp_1;\ ...\ ;\ vp_n]$ where id is the entity identifier and vp_i $(i = 1,\ \ldots\ ,\ n)$ the attribute or components values of e. These values may be atomic or constructed. They also may be entity identifiers or sets of entity identifiers (representing the mono-valued and the multivalued components).

In the same way, a relationship a of type type ta is formally defined as a tuple $[id;\ vp_1;\ ...\ ;\ vp_n]$ with the same semantics. However, the values vp_i $(i = 1,\ \ldots\ ,\ n)$ must be atomic or constructed.

A facet f_j of an entity e or a relationship a is then defined by the homomorphism f_j such that :

$f_j(e) = f_j([id;\ vp_1;\ ... ;\ \ vp_n]) = [id;\ f_j(vp_1);\ ... ;f_j(vp_n)]$
and
$f_j(a) = f_j([id;\ vp_1;\ ... ;\ vp_n]) = [id;\ f_j(vp_1);\ ... ;\ f_j(vp_n)]$

Applying such an homomorphism to vp_i may return :

1. an error code if the type of vp_i is not defined within the facet f_j,
2. a null value if the type of vp_i is not yet instantiated in the database,
3. a stored constant value which may be an atomic value, a constructed value, an entity's identifier or a set of entity identifiers, depending on the type of vp_i,
4. a constant that results from the application of a mapping function.

A mapping function between two facets f_i and f_j is formalised by an homomorphism $G_{f_i \to f_j}$ defined by :
$G_{f_i \to f_j}([id;\ f_j(vp_1);\ ... ;\ f_j(vp_n)]) = [id;\ G_{f_i \to f_j}(f_j(vp_1));\ ... ;\ G_{f_i \to f_j}(f_j(vp_n))]$

Both facets and mapping functions are formalised by homomorphisms. The ideal situation happens when $G_{f_i \to f_j}$ in an isomorphism.

4.3 Mapping functions in presence of schema changes

This section defines mapping functions for the more significant schema changes. Our approach has been applied to a list of schema changes that can be found in [6]. This list has been defined considering a schemas and every data type in it as abstract data types. The list of changes includes the restructuring of the sub-type relationship together with various schema changes like changing the domain of an attribute, changing an entity type composition, adding or deleting an entity type or a relationship type and so on. In this paper we only consider the most common schema modifications. These changes have a real side effect on data instances and programs.

Attribute deletion or migration This change is shown in the figure 4. The attribute $T_1.a$, in the version V_1 of the DB schema, is first deleted and migrates to $T_2.a$ in the version V_2.

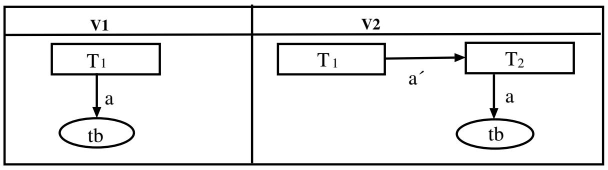

Fig. 4. Attribute migration

The mapping functions associated to such a change are defined by :
$G_{f_1 \to f_2}(f_1(T_1.a)) = f_2(T_2.a') = f_1(T_1).a'.a$ and $G_{f_2 \to f_1}(f_2(T_2.a)) = f_1(T_1.a)$
with $f_1(T_2.a) = f_2(T_1.a) = nil$

As an example, consider the migration of the attribute *Body.programer* to *Programer.name* (figure 1):

$G_{f_1 \to f_2}(f_1(b.programmer)) = f_2(p.name)$ and
$G_{f_2 \to f_1}(f_2(p.name)) = f_1(b.programmer)$ and
$f_1(p.name) = f_2(b.programmer) = nil$, where
m and b are instance identifiers of the entity types *Body* and *Module*.

That means that the message *b.programer* is rewritten into *p.name* in the all programs defined for the version V_1.

Restructuring the domain of an attribute Restructuring the domain of an attribute consists in transforming a domain from atomic values into tuples, or from tuples into sets, etc. As an example, the mapping functions associated with the restructuring of the domain of the attribute *Module.date_update* from a string to a tuple of strings (figure 1) are:

$G_{f_1 \to f_2}(f_1(m.date\_update)) = disaggregate(f_1(m.date\_update))$
$G_{f_2 \to f_1}(f_2(m.date\_update)) = aggregate(f_2(m.date\_update))$

We assume that $aggregate()$ and $disaggregate()$ are generic functions which transform an atomic value to an aggregation and conversely. So, if the attribute *Module.date_update* is physically stored as a string (according the version V_1) then the disaggregate functions is applied to it whenver it is accessed as a 3-uple.

Arbitrary transformation of an attributes' domain In some domain changes there is no mapping between the old and the modified domain. We call such a modification "an arbitrary domain change". For example, consider the transformation of the domain of the attribute *Body.size* from integer values to the set $\{"Low", "Medium", "Large", "Huge"\}$. In this case, the user must define an explicit function which retrieves a qualitative range for the attribute *Body.size* ("Low" etc.) given the integer value of their sizes. The reverse function does not exist, this case shows undecidable situations.

Transforming an entity type into a relationship type Transforming an entity type into a relationship type is quite specific to schema integration process. As an example, consider the *Module*'s transformation from an entity type to a relationship type (figure 1). In this case, there is no real restructuring needed because the path expressions which retrieve components are identical to those retrieving relationship participants. Mapping functions are only needed to translate the names of modules properties. They will be expressed by:

$G_{f_1 \to f_2}(f_1(m.spec)) = f_2(m.sp)$ and $G_{f_2 \to f_1}(f_2(m.sp)) = f_1(m.spec)$
$G_{f_1 \to f_2}(f_1(m.bodies)) = f_2(m.bod)$ and $G_{f_2 \to f_1}(f_2(m.bod)) = f_1(m.bodies)$

The following section gives some ideas leading to an effective implementation of our proposals.

5 Implementation issues

We consider the facets and mapping functions as basic services which may be used to implement all the strategies dealing with the impact of schema evolution on the existing databases and programs. Mapping functions may be used to just implement data conversion. They also may be used in order to use a same databse through several schema versions. A multifaceted data instance is then manipulated by polymorphic methods. Thus, for a same message selector we associate a set of method bodies (one body for each facet).

Let's assume the existence of a synthetic schema version according to which data is physically stored. To implement methods' polymorphism three cases must be considered :

i) a mapping function exists and can be automatically extracted. This case concerns especially all schema changes which consist in properties migration or domains restructuring. For such cases we assume the existence of a library of operation translator schemes. We then choose a translator scheme and instantiate its parameters which can be extracted from the specification of the schema change.

ii) A mapping f may be provided by the user. So, we replace the operation acceding to the property concerned by the user-defined procedure. This case is similar to the modification of $Body.size$ attribute from integer values to qualitative ones.

iii) We can not define a mapping. Thus, a data redundancy can not be avoided,

6 Concluding Remarks

Every database schema change may affect both data instances contained in the database and the programs operating on them. The two general strategies developed to deal with the impact of schema changes on instances and program consist in restructuring the database or allowing the coexistence of several schema versions into a single database. Using the database restructuring allows instances conformity to the modified schema. However, there isn't any strategy to transform programs in the same manner. Furthermore, the current approaches that use the strategy of databases restructuring, only consider the atomic changes of a schema which consist in attribute deletion or generalisation. In contrast, the coexistence of several schema versions in a same database may lead to programs adaptation by means of the emulation mechanism. However it is only used for some atomic changes. Such a mechanism causes also data redundancy.

We have proposed an approach consisting in a combination of both database restructuring and the emulation mechanism. Where each instance can be viewed as set of facets, each one representing its state in one schema version. Thus an instance can belong to more than one schema version. To eliminate data redundancy, only one facet is physically implemented, the others are derived by means of mapping functions. Such functions are also used to adapt programs to the schema changes.

References

1. Jose Andany, Michel Leonard, and Carole Palisser. Management of schema evolution in databases. In *17th International Conference on Very Large Data Bases (VLDB)*, 1991.
2. Gad Ariav. Temporally oriented data definitions: managing schema evolution in temporally oriented databases. *Data and Knowledge Engineering*, 6:451–467, 1991.
3. J. Banerjee, H.-T. Chou, H.J. Kim, and H.F. Korth. Semantics and implementation of schema evolution in object-oriented databases. *ACM SIGMOD conference, SIGMOD Record*, 16(3):311–322, 1987.

4. Philip A. Bernstein. Database system support for software engineering– an extended abstract–. In *9th Internat. Conf. on Software Engineering*, pages 166–178, 1987.
5. Nacer Boudjlida and Olivier Perrin. A formal framework and a procedural approach for data interchange. In *Third Int. Conf. on Systems Integration*, pages 476–485, Sao Paulo, Brazil, August 1994.
6. M. Bouneffa. *Gestion des Objects Complexes : Modélisation, Intégration et Évolution*. PhD thesis, Université Henri Poincaré (Nancy I), May 1995.
7. W. Kim and H.-T. Chou. Versions of schema for object-oriented databases. In F. Bancilhon and D.J. DeWitt, editors, *ACM SIGMOD Int. Conf. Very Large DataBases*, pages 148–59, Los Angeles, CA, 1988.
8. J.A. Larson, S.B. Navathe, and R. Elmasri. A theory of attribute equivalence in databases with application to schema integration. *IEEE transactions On Software Engineering*, SE-15(4), 1989.
9. E. McKenzie and R. Snodgrass. Schema evolution and the relational algebra. *Information Systems*, 15(2):207–232, 1990.
10. G.T Nguyen and D. Rieu. Schema evolution in object-oriented database systems. *Data and Knowledge Engineering*, 4(1):43–67, 1989.
11. S.L. Osborn. The role of polymorphism in schema evolution in an object-oriented database. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):310–317, 1989.
12. D.J. Penney and J. Stein. Class modification in the gemstone object-oriented dbms. *SIGPLAN Notices (Proc. OOPSLA '87)*, 22(12):111–117, 1987.
13. Amit Sheth and Vipul Kashyap. Sofar (schematically) yet so near (semantically). In D.K. Hsiao *et al.*, editor, *Interoperable Database Systems, IFIP Transactions A-25*. North-Holland, 1993.
14. Andrea H. Skarra and Stanley B. Zdonik. *Type evolution in an object-oriented database*, volume Research directions in object-oriented programming, pages 393–416. B. Shriver, Cambridge, MA, mit press edition, 1987.
15. Christian Thieme and Arno Siebes. Schema integration in object-oriented databses. In *Proc. of CAISE 93, LNCS 685*, June 1993.